15-323/15-623 Spring 2019

Project 3. Open Sound Control and O2

Due Feb 28

1 Overview

In this project, you will modify your program from Project 2 (or a new one) to accept control messages via the Open Sound Control (OSC) protocol. This is all leading toward Project 5, which is a live performance on computers with a small group, so you should keep that in mind as you design your Project 3.

Project 3 should result in an interactive music-making application. The interaction should consist of using TouchOSC (http://hexler.net/software/touchosc) as an interface to control a wxSerpent-based MIDI application. You should either use General MIDI for output or have a General MIDI mode to make it possible for the staff to try out your program and hear something reasonably close to your intended sounds.[1] Also, you should use a stock TouchOSC interface (there are many to choose from) rather than a custom interface. We encourage you to design your own custom interfaces, but please include a mode that uses a stock interface to make grading easier.

As with Project 2, your wxSerpent music program should implement a generative music algorithm. Since this project focuses on control, you should implement many more control capabilities than you did for Project 2. Here are some possibilities of parameters to control:

- Volume
- Density
- Tempo
- Color (plucked strings, bowed strings, woodwinds, percussion, etc.)
- Pitch region and range
- Scales/Chords/Keys/Harmony

In addition, you can add controls that control/trigger/vary any visual output that you designed as part of Project 2, or if you had no visual output and would like to add some, feel free to explore this path in Project 3.

2 About OSC

Open Sound Control is a simple protocol that is documented elsewhere (see the readings). Serpent source code has an option to include a direct Open Sound Control API[2], but the current release of Serpent (and wxSerpent) only includes the O2 API[3] (we'll also learn about that in class), so to use OSC, you will tell O2 to create a port for OSC messages and redirect these messages to an O2 service. Let's break this down into steps:

1. Initialize O2.
2. Create an OSC port and redirect OSC messages to an O2 service.

---

[1] (Note to self) Exam Question: Why does project 3 ask for General MIDI instead of plain ordinary MIDI? I.e., what problem does General MIDI address?

[2] See https://www.cs.cmu.edu/~music/serpent/doc/serpent.htm#open-sound-control

[3] https://www.cs.cmu.edu/~music/serpent/doc/serpent.htm#o2

3. Create an O2 service and add handlers.

2.1 Initialize O2

O2 requires a few things to run:

1. Initialize O2 by calling o2_initialize("appname", false), where "appname" is an application name – this must match the application name of any other O2 process with which you want to communicate, and *false* disables debugging information[4] Since you are currently only receiving OSC messages, the "appname" string can be anything, but it should not match an independent O2 process on the same network, or else the other application might "steal" messages intended for a local service. Choosing your initials will *probably* avoid any unintended communication. Choosing `"proj3"` is a bad idea.
2. Establish a clock by calling `o2_clock_set()`. Every O2 application[5] should have a master clock, and since this application will have only one (this) process, it should provide the clock.
3. Call `o2_poll()` to make O2 run. O2 runs in the same thread as everything else in your program, so you have to give it some CPU to run by calling `o2_poll()` frequently. You can use a scheduler and schedule these calls (you know how to make an event that reschedules itself), or even better, can simply set `sched_o2_enabled = true` which will enable an automatic call to `o2_poll()` inside `sched_poll()`.[6]

2.2 Create an OSC port and redirect OSC messages to an O2 service

Create a port for incoming OSC messages using:

`o2_osc_port_new(`*`service, port, false`*`)`

where *service* is a string, e.g. `"arpeggio"` or `"melody"`. A *service* is an O2 message destination address. The *port* parameter is a port number. Try something in the 8000 to 9000 range. Note that the port must not be allocated by some other program, e.g. Wikipedia says port 8000 is used by iTunes radio and other Internet radio stream protocols. There are no reserved ports for OSC comparable to port 80, which is reserved for HTTP. The last parameter is *`tcp_flag`*, which is *false* to indicate you want a UDP connection. (Because TouchOSC uses UDP.) Note that you will have to manually configure TouchOSC with the IP address of your computer and the *port* number given to `o2_osc_port_new`.

2.3 Create an O2 service and add handlers

Now, incoming OSC messages will be converted to O2 messages. For example, if TouchOSC sends the message `/1/fader1 0.3145`, and you gave service name `"melody"` to `o2_osc_port_new`, then O2 will convert the incoming message to `/melody/1/fader1 0.3145`, and send it via O2.

Therefore, you want to create a local service named `"melody"` and provide one or more handlers for it. You have the option of installing one handler for all messages to the service or a

---

[4] which in wxSerpent probably does nothing anyway – it's there so that some automatic debugging information can be generated in the future.

[5] An O2 *application* is not necessarily a single program or process. An O2 application consists of all processes connected by a network and running O2 that have matching *application names*.

[6] Search serpent/lib/sched.c for `sched_o2_enabled` to see how this works.

separate handler for each address. If you have a separate handler for each address, you can do full type checking on the incoming message and provide a specific handler for each address, so we'll take that approach.

Create the O2 service (assuming the service name is `"melody"`):
`o2_service_new("melody")`, where `"melody"` is the service name.

Register a handler for each address of interest:
`o2_method_new("/melody/1/fader1", "f", 'fader1_handler', true)`, where

- The first parameter is the address including the service name,
- The next parameter is the type string. Most if not all TouchOSC messages send a single floating point number (even push buttons), so the type string will normally be one "f" meaning one float.
- The handler: a Serpent symbol that names a Serpent function (see below).
- The last parameter is the *coerce* flag: It says that if you get something other than the types in the type string, O2 can *coerce* the values into the expected types. The handler will *always* get the expected types if you provide a type string. If *coerce* is false, then incoming integers or doubles or 64-bit integers will be rejected because of mismatched types (the message is simply ignored). This doesn't matter much for TouchOSC, except that if you wanted the parameter to be an integer (maybe 0 or 1), then you could set the type string to "i" and set *coerce* to true, and O2 would do the conversion for you.
- Be sure to register a handler for *each* address you want to receive. In some cases, it might make sense to write a loop, e.g. for `/melody/1/fader1`, `/melody/1/fader2`, `/melody/1/fader3`, etc. Also, you can use the same handler function for multiple addresses (although you will then probably need to make the behavior of the handler depend upon the address, which is passed as a parameter to the handler.)

Define the handler function, for example:

```
def fader1_handler(time, address, types, float_param)
    display "fader1_handler", time, address, types, float_param
```

The parameters are: *time*, the timestamp from the O2 message (but TouchOSC does not send timestamps, so you will get the *receive* time rather than the *send* time; *address,* the address string of the O2 message; *types*, the type string of the actual parameters (may differ from the message type string if coercion took place); followed by a parameter for each data parameter in the message. Because this handler is only registered with type string "f", we know there will always be one parameter of type float (but in Serpent, all floats are coerced to doubles, which Serpent claims are of type `'Real'`, and all int32 and int64 items are coerced to Serpent `'Integer'`[7].)

[7] 14 bits of 64-bit words are needed for type tags, so ints in Serpent are 50-bit signed 2's-complement integers that range from 2^{-49} to 2^{49}-1. And in case you are wondering, Serpent `'Real'` values are IEEE standard double-precisian floats that use all 64 bits. How is this possible? There are enough NaN ("Not a Number" – unused bit pattern) values in the IEEE representation to squeeze in all the other Serpent types: 2^{50} integers, 2^{48} addresses (object pointers), 2^{48} short strings, etc., including a few more bits as type tags. This technique is sometimes called "NaN boxing." Probably the most common use of NaN boxing in language implementations today is in JavaScript.

3 Configuring TouchOSC

You will have to configure the TouchOSC (client) with an IP address for your application (server) and use WiFi to connect. Note that if you are using a cable modem or DSL connection, you are on a local network created by your router/modem, and your local IP address is *translated* by your service provider when you communicate with other networks (the Internet). Thus, at home, if I go to whatsmyip.org, it says my IP address is 72.95.163.241, which is assigned by Verizon, but my local IP address is really 192.168.1.39. If I send OSC packets to 72.95.163.241, the packets will not be delivered.[8]

How do you find your local IP address for TouchOSC? Please see the TouchOSC documentation (http://hexler.net/docs/touchosc-appendix).

4 Additional Considerations

TouchOSC costs about $5 and runs on iPhone/iPad/iPod Touch/Android. You are expected to buy a copy (or if you really need to save $5 maybe you can work with a partner). You are *not* expected to buy an iPhone/iPad/iPod Touch/Android device. If you have nothing that will run TouchOSC, and it is inconvenient to borrow a device for testing, you can use DontTouchOSC – an application written in wxSerpent, which offers 4 buttons, 4 check boxes, 4 sliders, and an XYZ mouse input (Z = mouse down or up). DontTouchOSC sends OSC commands, but it is not a TouchOSC emulator, so the messages are different. You can extend DontTouchOSC with more controls if you wish, but then you'll need to submit your changes so we can run it with your program. DontTouchOSC sends to port 8001 of "localhost" by default. Run `donttouchosc.srp` using `wxserpent`.

Another program in the download for this project is `osc_monitor.srp` that you can run with `serpent` – this program has two modes. If you invoke it with the option "d", i.e.

```
serpent64 osc_monitor d
```

then the program will install handlers to receive each of the DontTouchOSC messages and print them for your inspection. (This may also work with at least some of the TouchOSC messages.)

If you run without the "d" option, i.e.

```
serpent64 osc_monitor
```

then the program will listen to *any* OSC message sent to its port, which is 8001 by default, and print a description of the message. This might be useful for debugging your TouchOSC setup, and you can also use `osc_monitor` as a guide to using O2 to receive OSC messages – a key part of this project. E.g. if you follow directions but your program does not receive OSC messages, try `osc_monitor`. If it *does* receive OSC messages, then at least you know something is wrong with your code and you have working code to study.

---

[8] Why not? The message would be routed by my WiFi router to Verizon, which would then turn it around and send it back to my router, but to deliver the message, my router would have to manipulate the packet, changing 72.95.163.241 to the local network address 192.168.1.39. But, mainly for security reasons, the router will only do this for "reply" messages, e.g. TCP connections to web servers. The unexpected OSC message is not a reply to anything and it will be dropped by the router.

> An aside: Speaking of debugging, we have some work in progress on a browser-based O2 monitor that can tap into O2 applications and spy on messages. Think of it as wireshark for O2. It includes a Javascript API for O2. If you are interested in working on this, let me know. - RBD

Your program should be documented with clear instructions of what to try, what the controls do, which controls are active, and what to expect. E.g. don't say "for a good time, move sliders 2 and 3;" instead, say "slider 2 controls the lowest pitch from C2 to C5, and slider 3 controls the note density from 1 to 20 notes per second."

Use many controls: at least 10.[9]

Controls should be effective. E.g. if a useful range of the AtomicSponge[10] parameter is 0.5-0.6, the full range of the slider should be mapped to the range [0.5, 0.6], and you shouldn't have to write instructions like "carefully keep the AtomicSponge parameter (slider 2) between 0.5 and 0.6 and really don't go to zero because that will cause a divide by zero error – ha ha, you lose." Try to make something you can perform.

Your program should correctly use timed MIDI messages. Open MIDI output with a latency of 10ms. Events in Serpent scheduled for time *t* should run at approximately time *t*, MIDI timestamps should correspond to exactly time *t* (rounded to ms), and due to the latency setting, the MIDI will be delivered fairly precisely at *t* + 10ms.[11]

Your program should use the programs/mididevice.srp code to allow midi devices to be selected from the menu.[12]

5 Extra requirements for 15-623 Students

Students taking 15-623 have additional requirements:

1. Your program should display all incoming OSC values that you use. You can write to `stdout` (using `print`, `display`, or `stdout.write`) or you can display to a `statictext` object in a wxSerpent window.[13]
2. As with Project 2, your code should have a drum machine or drum sounds as part of the algorithmic composition.

---

[9] Someone is going to ask: "If I implement a drum pad with10 drum sounds, does that count as 10 controls?" That would be disappointing. At least 4 or 5 truly separate dimensions of control (pitch, volume, tempo, randomness, etc.) are expected. In grading, we will consider there is a tradeoff: for the same amount of effort, you can have a fewer controls that do more interesting things or more controls that do simpler things, and we'll try to take that into consideration.

[10] I was trying to come up with an amusing parameter name and Atomic Sponge came to mind – not very funny, but hey, it's late, so I typed it into Google to see if something interesting came up, and I got >2M hits, including atomicsponge.com. Go figure.

[11] Maybe everyone will do this anyway. It's not a big challenge because the details are mostly hidden in the sched and midi-io libraries you are already using.

[12] Ditto.

[13] Note that "statictext" means "not user-editable," but the text can be changed under program control.

3. We will introduce an extra 10 points for general effort and quality – we really expect more from 623 students. So if you just barely do everything required, 323 students will get 100% and 623 students will get 90%.

6 Grading Criteria

1) Correctness: Does the program compile, run, and produce the desired behavior? Some key requirements are:
    a. At least 2 voices (see Project 2)
    b. Make it interactive; do not schedule far ahead, but you can compute one or two measures ahead if you are using templates or some approach that has "natural" units of computation more complex than just the next event.
    c. Provide tempo control.
    d. At least 10 parameters under OSC control. (see previous footnote about this)
    e. Proper use of scheduling for precise timing; no accumulated timing error or loss of synchronization.

2) Modularity: Your program should isolate different concerns. In particular, you should clearly separate (1) MIDI I/O, (2) the Graphical Interface, (3) data input (if any), (4) the implementation of application-specific real-time behavior (sequencer, drumming, or composition), and (5) the handling of OSC and O2 messages. These are not required to be in separate files.

3) Programming style: Code should be clearly written and commented to optimize readability (include high-level design and specifications at the top of the file or in a separate document, give concise comments within the code, avoid verbose or redundant inline comments.) A program with 10 lines of comments needs work. Low-level comments on every variable and method are not a substitute for quality documentation. No tabs in code. No very long lines; 80 characters is a good limit. Make sure the course staff can understand your program. Follow style guidelines in https://www.cs.cmu.edu/~music/serpent/doc/serpent.htm#style.

4) Instructions: A 2-page instruction manual might be about right for this project. At least we want to be able to read what to do and how it works. Don't write a book, but don't leave graders in the dark.

7 Hand-in Instructions:

You should put all your work, Source files and Documentation, into a ***zip*** file and submit it through Autolab.

Be sure that you include:

- Instructions on how to run and demonstrate everything you have implemented
- The specific control interface to use – either a built-in control panel *and its name* from TouchOSC (and tell us what device you used to run TouchOSC), DontTouchOSC, or a custom interface (submit the file and tell us the file name in the instructions).

Submit a **zip file**, no tar, gz, rar, or other file formats please. Autolab will automatically apply the extension ".zip" which makes all other formats difficult to open.